1 Objectives

The purpose of this lecture is to give you an introduction to higher order functions. As a result of this lecture, you should...

- know how to create higher order functions
- understand the concept of *closures*
- know how to use and write the following kinds of higher order functions:
  - function combination — `twice`, `compose`
  - interface changes — `curry`, `uncurry`
  - list processing — `fold_right`, `map`, `zip_with`

2 Examples

```
# let double x = x * 2;;
val double : int -> int = <fun>
# let inc x = x + 1;;
val inc : int -> int = <fun>
# let compose f g = fun x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

2.1 `map` and `fold`

```
# let rec map f lst = match lst with
    | [] -> []
    | x::xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map inc [2;3;4];;
- : int list = [3; 4; 5]
# let rec zip f alst blst =
    match alst,blst with
    | [],_ -> []
    | _,[] -> []
    | a::aa, b::bb -> f a b :: zip f aa bb;;
val zip : ('a -> 'b -> 'c) -> 'a list -> 'b list
          -> 'c list = <fun>
# let rec fold_right f lst z = match lst with
    | [] -> z
    | x::xs -> f x (fold_right f xs z);;
val fold_right : ('a -> 'b -> 'b) -> 'a list
                -> 'b -> 'b = <fun>
# fold_right (+) [2;3;4] 0;;
- : int = 9
```

2.2 `curry` and `uncurry`

```
# let curry f a b = f (a,b);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

```
# let uncurry f (a,b) = f a b;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

3 Problems

Here are some practice/study problems. We might do some of these in class.

1. What is a higher order function?
2. Write the function `twice : ('a -> 'a) -> 'a -> 'a` function, that takes a parameter `f` and a victim `x`, and applies `f` to `x` two times.
3. Write the `thrice` function.
4. Write `curry` and `uncurry`.
5. Write `flip : ('a * 'b -> 'c) -> 'b * 'a -> 'c`.
6. Write a function `flipuc` that flips uncurried functions. Do this using only `flip`, `curry`, and `uncurry`.
7. Write the function that has the type
   `('a -> 'b) -> 'a * 'c -> 'b`
8. Write the code for `map`.
9. What are the properites of a mapping recursion?
10. Using `map`, write a function that increments every element of a list.
11. Using `map`, write a function that doubles every element of a list.
12. Write the code for `fold_right`.
13. What are the properties of a folding recursion?
14. Use `fold_right` to write a function that takes a list and then returns it.
15. Use `fold_right` to write a function that sums up a list.
16. Use `fold_right` to write a function that takes the product of a list.
17. Use `fold_right` to write a function that takes the minimum of a list. Assume that every list has at least one element.
18. Using `fold_right`, write a function that increments every element of a list.
19. Using `fold_right`, write a function that doubles every element of a list.
20. Use `fold_right` to write a function that takes a list and removes all elements less than zero.
21. Use `fold_right` to write `map`.
22. Why can't `map` be used to write `fold_right`?